Reversivity, Reversibility and Retractability

Nikolai N. Nepejvoda

Program Systems Institute of RAS, Pereslavl-Zalessky, Yaroslavl Region, Russia, 152020
nepejvodann@gmail.com,
WWW home page: http://site.u.pereslavl.ru/Personal/NikolaiNNepeivoda/

Abstract. Three essentially different but usually mixed notions of program invertibility are considered. Reversivity when each action has a full inverse. Reversibility when each action can be undone (right inverse). Retractability when erroneous result can be retracted down to error in data. Constructive logical, algebraic, programming and realization aspects are considered for those types of programs.

Keywords: constructive logics, reversible computing, reversive logic, program inversion

1 Origins of reversivity

R. Landauer [1961] pointed out that there is an important kind of computations not investigated earlier and not intensively studied now (though not forgotten). Consider, for example, a superconductor based computer. Its elements are to be cooled by liquid hydrogen to provide superconductor properties of elements. Hot pollution could be fatal for this kind of processor. Thus we need all actions of our computer not to induce heat pollution. Landauer showed that it is physically possible if all actions are *invertible*.

The following formula describes the state of arts here:

Generalized Landauer – von Neumann principle

$$E_{diss} \geqslant T \times k_B \times \ln P \tag{1}$$

Here T is the temperature in K°, k_B is the Boltzmann's constant, P is the number of states of atomic computing element (assuming that energy needed to transfer between any two states is equal). A fresh work on this topic which contains an attempt of experimental verification of Landauer's principle is [2012L].

Bennett [1973] proposed in 1973 to make logically reversible computer (in the sense of boolean logic). He at first time made a big error which was inherited by further works.

In the common sense invertible action is that which can be wholly undone. For example adding block of text to Word file we always can delete it. This is semi-invertible action: we can undo it after it is done but cannot prevent it by doing an inverse action before it. For example accepting and rejecting changes in

Word document is a semi-invertible action (inside one Word session). Accepting and rejecting them in 'track changes' mode is a fully invertible one. This shows a deep difference between reversive (invertible) and reversible (semi-invertible) computing. The second one cannot diminish heat pollution and cannot be applied if our basic computing units are mostly invertible (e.g. quantum or DNA).

Toffoli [1980)] and Fredkin [1982] showed that it is possible to perform the full set of boolean computations on superconductor based computations almost without hot generation by the cost of doubled memory. A superconductor elements based cell can change state without hot generation if numbers of zeros and ones do not change. Thus Toffoli proposed to consider computations where each cell contains the same number of zeros and ones forever but they danced inside during computation process. Inevitable hot generators are only setting the initial state of a computer and reading the final result from some parts of superconductor cells.

Merkle [1992] proposed another variant of reversible binary logic devices. His goal is to beat physical barrier for computer productivity. His description of a problem is brilliant [1996]:
"If the exponential trends of recent decades continue, energy dissipation per logic operation will reach kT (for $T = 300$ Kelvins) early in the next century. Either energy dissipation per logic operation will be reduced significantly below $3 \cdot 10^{-21}$ joules, or we will fail to achieve computers that simultaneously combine high packing densities with gigahertz or higher speeds of operation. There are only two ways that energy dissipation can be reduced below $3 \cdot 10^{-21}$ joules: by operating at temperatures below room temperature (thus reducing kT), or by using thermodynamically reversible logic. Low temperature operation doesn't actually reduce total energy dissipation, it just shifts it from computation to refrigeration. Thermodynamically reversible logic elements, in contrast, can reduce total energy dissipation per logic operation to $\ll kT$."

The main error here is that almost all authors forget that to get a new memory location is also an energy-dissipating action. Moreover those who in concrete considerations do not allow this action (e.g. Merkle, Toffoli) do not mention this restriction in their comments.

There is a spread (rather little but slowly growing until this day) of works considering "reversible logic" and reversible computing. Too narrow point of view of *all* these works is to consider reversible computing only as boolean reversible transformations. Thus brilliant but restricted ideas of Merkle, Toffoli and Fredkin are accepted as "axiomatic" which cannot be discussed but only developed (as a good example see [2003]). We emphasize that boolean elements are only a *tradition* of current hardware but not its *sine qua non* property. Thus those aspects of reversivity which will become crucial if reversive super processors would be made technically: "How to program those exotic units?" and "What we cannot put into them without destroying their potentially best sides?" are not touched and even not seen.

And we are to emphasize the following:

REVERSIVITY

To overcome the Landauer principle we need full invertibility, not only possibility of undoing: each action f must have an action f^{-1} s.t. $f \circ f^{-1} = f^{-1} \circ f = e$ where e is an empty action.

This was assumed in the original works and repeatedly pointed out by independent researchers (see e.g. [2001Bub,2005Mar,2012L])

Thus *we need more general mathematical and more logical consideration of this extremely interesting and specific domain.* Constructive logic of reversivity is the first step into a new realm.

Now we consider three notions of (semi) invertibility in the order from an almost traditional up to the most striking, leading to a fresh paradigm and therefore totally inadequately treated.

2 Retractability

Main peculiarity of constructive logics is that they are not truth-value based. Formulas are treated as problems and we are interested in solution of this problem which can be for different logics and theories whether a mathematical object or a program or another entity. If a is a solution of a problem represented by a logical formula A we say 'a realizes A' (a ® A).

I don't know a systematic survey of all branches of constructive mathematic written in English. Constructivism now is a system where different main resources and different resource restrictions lead to very exotic, mutually inconsistent and fine systems (see [2011NNN]). The logic of knowledge is the intuitionistic logic; the logic of money is the linear logic of Girard; the logic of time, automata and real actions is the nilpotent logic; the logic of reversivity and soul is partially described below. One of main philosophical consequences of constructivism is that it is a mortal trick of a society to allow those people which are thinking inside the logic of money to govern the real things or the knowledge discovering process.

Some formulas in constructive logics can have a trivial underlying problem and thus be treated as descriptive ones (usually as classical). For example formulas $\neg A$ in intuitionistic logic are descriptive and classical logic can be isomorphically embedded into intuitionistic (but not vice versa).

In this section we use almost forgotten and not developed further results of constructive logic applications to programming. "*Sturm und Drang*" of this topic was in last 70-ths — early 80-ths. There were many fine algorithms and strong results (see e. g. [NNN1982,Mar1982,1991,1998]) but their practical applications meet some obstacles.

Obstacle 1. Swamp. As usually, after big advance there arose a huge boring work to develop techniques and technologies of practical work with: new possibilities opened (it is easier but often demands a full change of traditional manner of actions); with new dangers and shortcomings which accompany each innovation. Big benefits always go together with big disadvantages. To lead new

footpath through a swamp is hard and boring work. We cannot make here big promises (if we didn't lost remains of honor). We cannot show any spectacular results here. We cannot easily explain significance of our work to outsiders (= peer reviewers). Thus we cannot get grants and our work dies before reaching the another bank of a swamp and a new living space. Constructive logics had faced with the necessity to develop essentially another technique of formalization because traditional one did not works good. This work had become their swamp.

Obstacle 2. Disorientation. The theoretical works are mostly disorienting here because they usually treated so "practical and important" examples as factorial or Ackermann's function. These examples are both too primitive in their structure and too connected with the specific data types and the traditional set of atomic operations but easily understood by theoristsoutsiders. They were out of mainstream of both theoretical and practical informatics which now works with abstract and varying structures and constructs new programs not from the computer primitives but from modules, objects and patterns.

Obstacle 3. Fake advertising. Japan 'Fifth generation project' used constructive logic as one of its banners. In reality they have adopted more traditional but leading into deadlock tools (e.g. Prolog). A reputation of a fallen project was transferred into its 'used' theory.

A program is *retractable* if it allows to retract from properties and/or errors in the result to properties or errors in the data. The logic of retractable structured program is symmetric intuitionistic logic (SIL) investigated by I. Zaslavsky [1979].

In this logic there are only constructive connectives $\Rightarrow \vee \& \sim \forall \exists$. Their semantic is defined through the two notions of realizability: positive and negative one.

Definition 1. *Realizabilities for SIL.*

1. $\langle a,b\rangle \; ®^+ \; A \,\&\, B \equiv a \; ®^+ \; A \wedge b \; ®^+ \; B$;
   $\langle i,c\rangle \; ®^- \; A \,\&\, B \equiv (i = 1 \wedge c \; ®^- \; A) \;$ *or* $\; (i = 2 \wedge c \; ®^- \; B)$;
2. $\langle i,c\rangle \; ®^+ \; A \vee B \equiv (i = 1 \wedge c \; ®^+ \; A) \;$ *or* $\; (i = 2 \wedge c \; ®^+ \; B)$;
   $\langle a,b\rangle \; ®^- \; A \vee B \equiv a \; ®^- \; A \wedge b \; ®^- \; B$;
3. $\langle f,g\rangle \; ®^+ \; A \Rightarrow B \equiv$ *for all* $a \; (a \; ®^+ \; A \supset !(a\ f) \wedge (a\ f) \; ®^+ B) \wedge$
   for all $b \; (b \; ®^- \; B \supset !(b\ g) \wedge (b\ g) \; ®^- \; A)$;
   $\langle a,b\rangle \; ®^- \; A \Rightarrow B \equiv a®^+ \; A \wedge b®^- \; B$;
4. $a \; ®^+ \sim A \equiv a \; ®^- \; A$;
   $a \; ®^- \sim A \equiv a®^+ \; A$;
5. $f \; ®^+ \; \forall x \, A(x) \equiv$ *for all* $a \; (a \in U \supset !(a\ f) \wedge (a\ f) \; ®^+ \; A(a))$;
   $\langle u,a\rangle \; ®^- \; \forall x \, A(x) \equiv$ *exists* $u \; (u \in U \wedge a®^- \; A(u))$;
6. $\langle u,a\rangle \; ®^+ \; \exists x \, A(x) \equiv$ *exists* $u \; (u \in U \wedge a®^+ \; A(u))$;
   $f \; ®^- \; \exists x \, A(x) \equiv$ *for all* $a \; (a \in U \supset !(a\ f) \wedge (a\ f) \; ®^- \; A(a))$;

Here $!t$ denotes «value of t exists»; U is the set of all primitive objects of our model.

The usual rules for negation are valid for SIL. There is an extraction algorithm which can extract two procedures from a proof of formula of the form:

$$\forall x_1 \ldots x_n \, (A_1 \& \ldots \& A_m \Rightarrow \exists y_1 \ldots y_k \, (B_1 \& \ldots B_l))$$

The first procedure finds y for all x satisfying A. The second one shows how to find such j that $\sim A_j(x_0)$ having $\sim B_i(x_0, y_0)$. Thus we have both a program and a routine to analyze its errors.

Now we consider an example. Let in a subtheory (essentially constructive formulas are specified by their realizations)

$$\begin{array}{l}\forall x\,((A(x) \Rightarrow N(x)), \qquad \varphi \circledR \forall y\,(N(y) \Rightarrow \sim \exists x\, M(x)),\\ g \circledR \forall x(C(x) \Rightarrow L(x) \vee E(x) \vee M(x)),\\ \forall x\,(L(x) \Rightarrow D(x)), \qquad \forall x\,(H(x) \Rightarrow T(x, (x\ f)))\end{array}$$

which is a part of a constructive theory describing some packages of programs we proved a formula

$$\forall x\,(A(x)\ \&\ (\forall x\,(C(x) \Rightarrow D(x) \vee E(x)) \Rightarrow \exists y\, H(y)) \Rightarrow \exists y\, H(y)) \Rightarrow \exists z\, T(y, z))$$

by the following proof:

$$\begin{array}{l}* \ A(z),\ \forall x\,(C(x) \Rightarrow D(x) \vee E(x)) \Rightarrow \exists y\, H(y),\ z \text{ is arbitrary}\\ \quad N(z)\\ \quad \sim \exists x\, M(x)\\ \quad * \ C(u),\ u \text{ is arbitrary}\\ \quad\quad L(u) \vee E(u) \vee M(u)\\ \quad\quad \sim M(u)\\ \quad\quad * \ L(u) \qquad * \ E(u)\\ \quad\quad \mid D(u)\\ \quad \forall x\,(C(x) \Rightarrow D(x) \vee E(x))\\ \quad H(c_1)\\ \quad T(z, (c_1\ f))\end{array}$$

$$\begin{array}{l}* \ \sim T(y, z),\ y, z \text{ are arbitrary}\\ \quad \sim A(x) \vee \sim (\forall x\,(C(x) \Rightarrow D(x) \vee E(x)) \Rightarrow \exists y\, H(y))\\ \quad * \ \sim (\forall x\,(C(x) \Rightarrow D(x) \vee E(x)) \Rightarrow \exists y\, H(y))\\ \quad\quad \sim H(x),\ x \text{ is arbitrary}\\ \quad\quad \exists x\,(C(x)\ \&\sim D(x)\ \&\sim E(x))\\ \quad\quad L(c_2) \vee E(c_2) \vee M(c_2)\\ \quad\quad \sim L(c_2) \qquad \sim E(c_2) \qquad * \ \sim A(y)\\ \quad\quad M(c_2)\\ \quad\quad \sim N(y)\\ \quad\quad \sim A(y)\\ \quad \sim A(y)\end{array}$$

Here our direct program is

$$\begin{array}{l}\Phi: \ \mathbf{func}\ (\mathbf{obj}, \mathbf{func}(\mathbf{func}(\mathbf{obj})\mathbf{void} \oplus \mathbf{void})\ \mathbf{obj})\ \mathbf{obj} =\\ \lambda x, \Psi.\,((\lambda x.\,\mathbf{case}\ (x\ g)\ \mathbf{in}\ 1:1, 2:2, 3:\ \mathbf{error}\ \mathbf{esac}\ \ \Psi)\ f)\end{array}$$

If its result is wrong, an error is in A. The reason of this trouble is probably a wrong value of x which formally does not enter into a resulting program.

A procedure of backward analysis given originally for error diagnosis can be used for other kinds of backward computations of program condition. We have to stress that during this kind of backward computations we are interested not in restoring of values but in information about initial values which had been lost or not taken into account before program computation.

Retraction is first order process even for functional programs in the majority of practical situations (roughly speaking if our positive suppositions do not include a demand to grant an erroneous result of a procedure).

Moreover here we have an interesting duality. G. S. Tseytin pointed out in 1970 that program values are not sufficient to analyze a program. Program is surrounded by *ghosts* which are necessary to understand and to transform a program but are at least useless during its computation. During retraction ghosts become computable entities while values of direct program become ghosts.

3 Reversibility

Invertibility cannot be considered as a property of some exotic classes of hardware computations. In business and legal practice it is sometimes necessary to provide a possibility to restore easily the precise state of the whole system for *each* given moment in the past. In many interactive program systems it is necessary to provide unrestricted undoing.

Program allowing unrestricted undoing is called *reversible*.

Definition 2. *Let X be an enumerated set. Let $\mathfrak{C}(X, X)$ be a set of all total computable functions $f : X \to X$. A semigroup $R \subset \mathfrak{C}(X, X)$ having a neutral element $e = \lambda x.x$ and having a right inverse f^{-1} for each f (i.e. such f^{-1} that $f \circ f^{-1} = e$) is called* reversible computability *upon set of objects X.*

We emphasize once more that reversibility has no relation to problem of heat generation and programming physically reversive units. Nevertheless it is practically essential and interesting.

Main results on ≪possibility to make any Turing computable function invertible≫ consider only reversibility. In publications two notions ≪reversivity≫ and ≪reversibility≫ are systematically muddled together.

A reversible processor can in principle work autonomously. But it is necessary to remember lower bounds of extra resources needed for reversibility as stated in [1973].

$$\text{Time} > 3^k \cdot 2^{O\left(\frac{T}{2^k}\right)} \qquad \text{Store} > S \cdot (1 + O(k)) \tag{2}$$

where k can be chosen between 1 and $\log_2 T$. Here we have somewhat shifted notions. Bennett result and its further generalizations consider a problem how to simulate an irreversible computation on reversible processor. In practice we are interested in the same result but not in the same computation flow. Thus this bound is theoretically correct but practically somewhat misleading.

Moreover we don't need to assure undoing down to atomic actions in reversible computing because reversibility is needed only for external reasons (say many legal and business program must be able to reconstruct the state of the system for any previous time moment). Hence *a reversible program can use modules written in irreversible manner if we grant undoing of their results.*

From this point we can see strategic mistakes made in the design of reversible language Janus [2007]. For example, there is a brilliant invention of Janus authors that each unary function f is extended up to its reversible extension

$$(x\ y\ g) = \langle x * (y\ f), y\rangle$$

where $\forall x, y, z\,(x * z = y * z \supset x = y)$. They showed that each unary function can be extended in such manner. But this excellent shot had a wrong goal and is missed. Of course it is too much for reversibility but too less for reversivity (it grants only undoing).

Now we will outline some reasons why there is no need of a reversible programming languages. Reversibility can be reached by a discipline of programming in a traditional structured language.

It is necessary to prohibit completely any invisible side effects of functions and procedures.

Each procedure f can have only **in** and **inout** parameters and have a dual procedure undo_f which makes undoing of its effects.

Each function can have only **in** parameters and needs no undoing procedure.

Let us remember the remark of E. Dijkstra and D. Gries [2010Dij,1981Gr] that natural assignment statements are to have a form $x_1, \ldots, x_n \leftarrow t_1, \ldots, t_n$.

Let now classify all variables in all program points as virgin, used and monk ones. Now each assignment must have one of two forms. A multi form $x_1, \ldots, x_n \leftarrow y_1, \ldots, y_n$ where the right side contains no virgin variables and each used variable from the left side must occur at least once in the right side. A single form $x \leftarrow t$ where x is virgin. After this x becomes used.

Case statements and loops are unified in a spider statement form proposed in [NNN1982] (this is an extension of Dijkstra's loops and guarded commands and had used e.g. by Bel'tyukov in his language КЫЛ):

for i,x1,. . . ,xn **do** N1: S1,. . . , Nk: Sk **out** L1: T1,. . . , Lm: Tm **new** U **od**.

Here i is an integer variable, Nj, Lj are integers, $i, x1, \ldots, xn$ are frozen inside each loop step and can be changed only in **new** part. Inside loop they are called *monks*.

A spider loop is executed as follows. If the value of i does not equal to any of Nj, Lj there is an error. If it is equal to some Nj, the corresponding statement Sj is executed, then U is executed and the loop continues. If it is equal to some Lj, the loop is finished after execution of Tj and U.

Those conditions (easy to formulate as a discipline and technology of programming and easy to check) are sufficient to grant reversibility.

Thus conditionals and loops do not hinder reversibility and only force us sometimes to introduce some additional information.

Constructive logic of reversibility is a good problem for a new research.

4 Reversivity

Constructive reversive logic (CRL) was described and investigated in [2009]

For a mathematical semantic we consider an arbitrary group G. One more important step was proposed and successfully developed by J.-Y. Girard in his linear logic (using commutative monoid to represent money-spending actions). For our case it sounds as follows:

States are the same group as actions.

Thus G is called both *the group of actions* and *the group of states*.

Each propositional letter corresponds to a subset of the group[1], and each element α of the group represents the function $\lambda x.\, x \circ \alpha$.

Thus in the functional language based on groups application of f to a is to be written (a f) in contrary to usual (f a). Composition of group elements $a \circ b$ can be understood by any of three ways:

1. We perform the state-transfoming action a then the action b;
2. We apply the function b to a;
3. We construct a composition of functions a and b.

All those interpretations are compatible and fully interoperable. This is the main peculiarity of group as a space of elements and actions.

CRL is a propositional logic. The primitives of reversive logic language are propositional symbols A, B, $C\ldots$, five connectives of classical logic ($\supset$, $\equiv$, $\wedge$, $\vee$, $\neg$) called here *descriptive connectives*, four constructive logical connectives $\Rightarrow, \&, \sim, E$. E is null-ary, $\neg$ and $\sim$ are unary, all others are binary. We adopt the following priority of binary connectives (form the weakest one up to strongest): $\Rightarrow$ & $\equiv\supset$ $\wedge\vee$ but we use parenthesis rather than priorities of $\equiv\supset$ and $\wedge\vee$ for the needs of easy reading. Let *signature* Σ be a nonempty set of propositional symbols.

Classical connectives are read and understood in standard way. $\Rightarrow$ reads "can be transformed", $A\&B$ reads "*sequential conjunction*" or "A then B"[2], $\sim A$ is a preventive negation which can be read in different contexts as "undo A" or "prevent A".

Classical and constructive connectives are fully interoperable[3] and can be mixed arbitrarily. This is not the case in other constructive logics of restricted constructions.

A formula A is *descriptive* if there are no constructive connectives. A formula A is *pure constructive* if there are no classical connectives. Therefore propositional letters are both classical and pure constructive formulas.

[1] Attention! This subset is not obliged to be a subgroup or stable in the sense of the linear logic. This is a principal distinction from quantum logics and other algebraic logics.

[2] Of course we can read this "and" in the sense of famous Kleene's examples: "Mary married and born a child", "Mary born a child and married".

[3] As it is called in modern programming.

Our main semantic notion is "element *a realizes* a formula A in an interpretation I" ($I \models a\circledR A$). If an interpretation is fixed we omit I.

$\triangleq$ means «is equivalent by definition»

Definition 3. Interpretation *of a signature Σ is a pair consisting of a group G and of a function $\zeta : \Sigma \to \mathbb{P}G$ which maps propositional letters into power set of G. A subset which is assigned to a propositional symbol A in I is denoted $\zeta_I(A)$. If I is fixed we omit the index.*

Definition 4. Realization *of a formula in the interpretation I.*

1. $a \circledR A \triangleq a \in \zeta(A)$ *where A is propositional letter and $A \in \Sigma$.*
2. *For all classical connectives their definitions are standard. E.g.* $a \circledR (A \wedge B) \triangleq a\circledR A$ *and* $a \circledR B$.
3. $a \circledR (A \Rightarrow B) \triangleq \forall b \in G\ (b \circledR A \supset b \circ a \circledR B)$. *Thus a transforms solutions of A into solutions of B.*
4. $a \circ b \circledR (A \,\&\, B) \triangleq a \circledR A \wedge b \circledR B$. *$A$ solution of B is applied to a solution of A.*
5. $a \circledR \sim A \triangleq a^{-1} \circledR A$. *$a$ undoes a solution of A or prevents it.*
6. $a \circledR E \triangleq a = e$.

The set of realizations for A is denoted $\circledR A$.

Whenever an interpretation I is mentioned it is assumed that I is an interpretation for the signature of our formulas.

Definition 5. A is true *in I if $\circledR A = G$. A* is valid *if A is true in each I. Validity of A is denoted $\models A$.*
A is realizable *in I if $\circledR A \neq \emptyset$. A* is totally realizable *if A is realizable in each interpretation I.*

CRL diverges from other constructive logics. Say, both $A\&A \Rightarrow A$ and $A \Rightarrow A\&A$ are invalid. A is realizable iff $\sim A$ is but these formulas do not imply one another. $A \Rightarrow A$ is totally realizable but is true iff A or $\neg A$ is true. Quantifiers can be expressed here on propositional level. For example

$$\forall A \equiv (A \vee \neg A) \Rightarrow A.$$

Here we have no constructive disjunction. If introduced it demands an «interleaving product» of groups: a group of all products $a_1 \circ b_1 \circ \cdots \circ a_n \circ b_n$ where a_i are from realizations of A and b_i are from one of B. This destroys finiteness and means that conditionals demand increasing memory. Analyzing constructions of Fredkin and Toffoli we see that it is.

So pure reversive programming language is to be without conditionals and loops but from the very beginning functional one [2009A,2009Izh,2009VIZ,2010]. In practice we are to use irreversive operations (at least initializing and result writing) and very restricted use of conditionals and loops. Of course there are no recursions and reversive language is not Turing-complete. Atomic computing elements for reversive computer are to be group-valued not binary.

A basic skeleton for reversive programming language can be the following.

Programs can be pure, conditional, cyclic and generic. Atomic actions are pure. Compositions of pure programs are pure. Functions with pure body are pure.

Pure programs are conditional. A construction

if P **then** t **else** r **fi**

is conditional if t and r are conditional. Compositions of conditional programs and functions with conditional bodies are conditional.

Pure programs are cyclic. A construction

to N **do** t **od**

is cyclic if t is cyclic and N is a constant natural number. Compositions of cyclic programs are cyclic. Functions with cyclic bodies are cyclic.

If pr is a program of each three above classes then -(pr) is a program of the same class. Brackets around atoms and functions can be omitted.

Generic program is a composition of programs of types above. Reversive module has a form
<definitions> <input> <generic program> <output>.
Forms of <definitions>, <input> and <output> parts we will not consider at the moment.

Consider an example of program scheme with one type only and without parameters of functions.

```
DEFINITIONS # All names used in a program are specified here
atom a1, a2, a3, a4, a5, a6, a7
predicate p1, p2
function f=(a1; if p1 then -a2; a3; a1 else a4; -a1 fi)
function g=(a1; to 51 do -a1 od)
function h=(a1; a3; -a1)
END DEFINITIONS
INPUT
   # initial values of all atoms and predicates are given here;
   # usually they are computed by external program
   # and transferred into
p1=¬(a4,a6) # if the domain of a predicate
   # or the value of an atom is fixed for all executions
   # it can be defined inside
   ...
END INPUT
-{to 14 do
   -g; h; a7;
   od; a2}; # we take an inverse of the whole program block
if p2 then -f; h else f fi
```

```
f; -g; -a4; h;
OUTPUT # a substructure transferred to external processor
   # is defined here
   ...
END OUTPUT
```

Thus conditional parts cannot be used inside cyclic parts and vice versa.

The problem of data types for reversive programs is fine and interesting. For example we cannot restrict ourselves by direct products of some standard groups. Consider an example. Let in a conditional statement **if** P **then** t **else** r **fi** functions t and r are computed dynamically. Then if G is a group of programs and H is a group of data we are to represent the corresponding group as follows. Its underlying set is $\mathbb{Z}_2 \times G \times G \times H$, and the group operation is

$$\begin{aligned} \langle z, a_1, b_1, c_1\rangle \circ \langle 0, a_2, b_2, c_2\rangle &= \langle z, a_1 \circ a_2, b_1 \circ b_2, c_1 \circ c_2\rangle \\ \langle z, a_1, b_1, c_1\rangle \circ \langle 1, a_2, b_2, c_2\rangle &= \langle z \oplus 1, a_1 \circ b_2, b_1 \circ a_2, c_1 \circ c_2\rangle \end{aligned} \tag{3}$$

Say, our program scheme with atoms from a group G needs much more complex group to be executed. The technique of computation of this group during translation of reversive program needs somewhat sophisticated algebraic technique and will be published in a separate paper. Here only note that

1. pure programs do not change a group;
2. each written loop adds an additive constant to the number of the group elements;
3. each executed conditional (roughly speaking) doubles the number of elements in a group.

During computation flow a group remains the same. Each change of a program can change its group.

Therefore a practical reversive processor can be a mill which makes a large amount of transformations with a low number of branches.

Each proof of $A \Rightarrow B$ in a reversive constructive logic gives a pure program to reach a state where B holds from any state with A. By the standard technique of precondition computation described in Gries [1981Gr] we can extend specifications of pure segments up throughout our program by conditions which hold in given points inside our programs. The whole process demands a very sophisticated and original logical tools and was partially described in [2009,2009Izh,2009VIZ].

So we see that realization of reversive programming demands a new theoretical and practical paradigm. It is to compose non-standard logical and programming tools together with algebraic ones into a single system. Underestimation of complexity and misunderstanding of nature of this problem are two main sources of 30-year stagnation in this domain, both in theory and in practice.

And last but not least. A reversive program must be reversive down to the atomic constructions and here we cannot use modules written in other languages.

Moreover reversive modules also cannot be used by other reversive programs because input and output destroy reversivity. We can use only function definitions given in a reversive language.

Applying our considerations of program retraction to the case of reversivity we see that the reversive constructive logic provides an instrument of effective retraction but this retraction is in its essence irreversive. Therefore error or condition analysis for a reversive processor must be performed by traditional one.

5 Conclusion

1. There are three substantially different but usually mixed notions of inverse computability. They need different tools and use different logics.
2. A reversive computation demands full invertibility of actions. Only it can grant minimization of heat pollution.
3. Reversive computability is not Turing-complete and a reversive processor can work only as specialized unit of an usual (for example von Neumann) computer.
4. Binary elements are maybe the worst choice for reversive computation. This process demands group-based elements.
5. It is necessary to compute in a reversive program the algebraic structures of data types and of the whole data space before program compilation because each modification of programs changes all data structures in it. This algebraic computation can be somewhat sophisticated.
6. A reversible computing (unrestricted undoing) can be implemented in traditional computers by traditional programming languages as a discipline of programming.
7. A program retraction (computation of precondition which hold or fail for the given result) can be made by means of almost traditional logic. During retraction values and ghosts are interchanged.

References

1961. Landauer. R: Irreversibility and heat generation in the computing process. IBM J. of R & D, 5, 183–191 (1961)

2001Bub. Bub, Jeffrey: Maxwell's Demon and the Thermodynamics of Computation. Stud. Hist. Phil. Mod. Phys., **32**, No. 4, pp. 569—579 (2001)

2005Mar. Maroney, O J E: The (absence of a) relationship between thermodynamic and logical reversibility. Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics. **36**, Issue 2, 355-–374 (2005)

1973. Bennett, C. H.: Logical reversibility of computation. IBM J. of R & D, 17, no. 6, 525–532 (1973)

1980). Toffoli, T. Reversible Computing. MIT TR MIT/LCS/TM-151 (1980)

1982. Fredkin, E. and Toffoli, T.: Conservative logic. Int.l J.l of Theor.l Phys., 21, 219–253 (1982)

1992. Merkle, R.C.: Towards Practical Reversible Logic. Workshop on Phys. and Comp., PhysComp '92, October, Dallas Texas; IEEE press (1992)
1996. Merkle R. C.: Helical logic. Nanotechnology, 7, 325–339, (1996)
2003. Shende, V. V. Prasad, A.K. Markov, I. L., Hayes, J. P.: Synthesis of Reversible Logic Circuits. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 22(6), 710–722 (2003)
2012L. Antoine Berut, Artak Arakelyan, Artyom Petrosyan, Sergio Ciliberto, Raoul Dillenschneider & Eric Lutz: Experimental verification of Landauer's principle linking information and thermodynamics. Nature, March 2012.
2007. Yokoyama T., Glück R.: A reversible programming language and its invertible self-interpreter. Partial Evaluation and Program manipulation. (2007)
2010Dij. Dijkstra, Edsger W.: A Discipline of Programming. Prentice Hall (2010)
1981Gr. Gries, D.: The Science of Programming. (1981)
2011NNN. Nepejvoda, N.N.: Уроки конструктивизма. Geidelberg: Lambert Academic Publishing, 98 pp. (2011)
2009. Nepejvoda, N.N.: Реверсивные конструктивные логики. Логические исследования, 15, 150—168 (2009)
2009A. Непейвода А. Н.: О сюрьективной импликации в реверсивной логике. VI Смирновские чтения по логике (2009)
2009Izh. Непейвода А. Н. Элементы реверсивных вычислений Управление большими системами труды VI всероссийской школы-семинара молодых ученых, Ижевск (2009))
2009VIZ. Непейвода А. Н.: О реверсивной альтернативе традиционным вычислениям. Трехмерная визуализация научной, технической и социальной реальности. Технологии высокополигонального моделирования : труды Второй междунар. конф., Ижевск (2010).)
2010. Непейвода А. Н.: Функциональное программирование над группой. Системный анализ и семиотическое моделирование: труды первой всероссийской конференции, 2011, Казань (2011)
1979. Заславский И.Д.: Симметрическая конструктивная логика. Ереван, (1979)
NNN1982. Nepejvoda N. N. Logical approach to Programming. Logic, methodology and philosophy of science VI. 109–122 (1982)
Mar1982. Martin-Löf P. Constructive mathematics and computer programming. Logic, methodology and philosophy of science VI. 153–179 (1982)
1991. Nepejvoda N. N. A bridge between constructive logic and computer programming. Theoretical Computer Science, **90** 253–270 (1991)
1998. Nepejvoda N. N. Some analogues of partial and mixed computations in the logical programming approach. New Generation Computing, **17**,309–327 (1999)